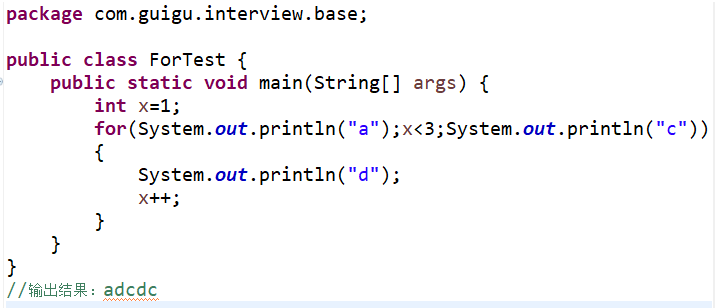
基础笔面试题

1.For循环测试



解析：

for循环分析

For( c1 ; c2 ; c3 )

{

循环体;

}

1、先执行初始化语句c1，此语句只在初始化的时候执行一次，此后不再执行

2、随后执行语句c2，判断循环条件是否成立，如果成立则执行循环体中的语句，如果不成立则跳出循环结束操作

3、随后执行语句3，控制循环条件

4、以此类推循环执行2、3操作，直到循环结束

由此可分析上述问题中先输出字符串“a”，随后判断“1<3”成立，执行循环体中的语句，输出字符串“d”，且x执行加1操作，随后输出字符串“c”，以此类推，继续判断x<3是否成立（2<3成立），执行循环体操作，输出“d”，x加1，此时x的值为3，继续判断循环条件，此时循环条件不成立，因而结束循环操作。

最后输出结果“a d c d c”

2.利用for循环控制打印下列图形

**package** com.guigu.interview.base;

**public** **class** TriangelTest {

**public** **static** **void** print1(**int** line)

{

**for**(**int** i=1;i<=line;i++) //控制行数

{

**for**(**int** j=1;j<=i;j++) //控制每行打印的字符个数

System.***out***.print("&");

System.***out***.println();

}

}

**public** **static** **void** print2(**int** line)

{

**for**(**int** i=1;i<=line;i++) //控制行数

{ //拆分为两种符号进行打印，空格和字符&

**for**(**int** j=1;j<=line-i;j++)

System.***out***.print(" "); //打印空格

**for**(**int** j=1;j<=i;j++)

System.***out***.print("&"); //打印字符&

System.***out***.println(); //换行

}

}

**public** **static** **void** main(String[] args) {

/\*\*打印三角形1（可根据指定行数打印）

\* &

\* &&

\* &&&

\* &&&&

\* &&&&&

\*/

System.***out***.println("三角形1：");

TriangelTest.*print1*(5);

/\*\*打印三角形2（可根据指定行数打印）

\* &

\* &&

\* &&&

\* &&&&

\* &&&&&

\*/

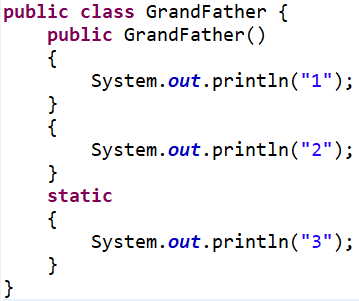
System.***out***.println("三角形2：");

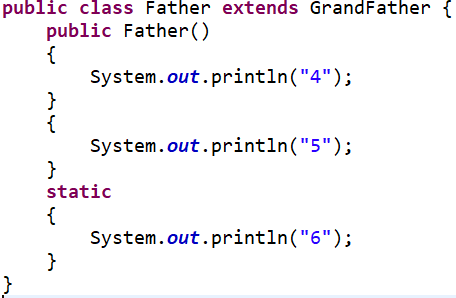
TriangelTest.*print2*(5);

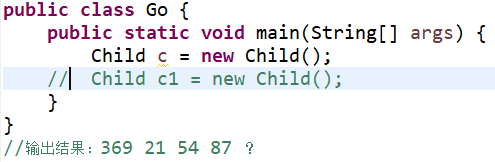
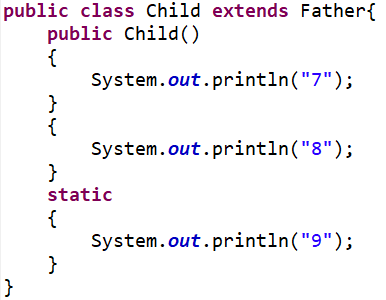
}

}

3.静态代码块







解析：

代码块是在类初始化的时候自动执行的，其优先于构造函数执行

* static初始化块则是在类加载的时候自动执行1次，若同一个java类中出现多个static初始化块则按照出现的先后顺序执行（只执行1次）
* 非static的初始化块语句会在类的对象创建的时候被调用1次
* 构造方法是在类对象被创建的时候执行
* 类加载的时候执行static语句块（只执行一次），对象创建的时候先执行非static语句块后执行构造方法，对象创建一次执行一次

程序执行的时候需要创建Child对象，则程序依次加载GrandFather、Father、Child类，其相应static初始化块依次执行，输出“3 6 9”，随后依次执行每个类中的非static初始化块、初始化构造方法，分别依次输出“21 54 87”

4.==与equals的比较

内存的五片区域 ：寄存器，栈内存，堆内存，方法区，本地方法区

栈内存：存储的都是局部变量，一旦使用完毕，立即释放

堆内存：存储的是实体(对象)

Java程序中有两种方式是两个变量是否相等。一种是利用==判断，另外一种是使用equals方法来判断。

当使用==进行判断的时候，如果两个变量都是基本类型的变量，并且都是数值类型，则只要两个变量的值相同时，便返回true。但如果两个变量的类型是引用类型，则只有两个变量均指向相同的对象才返回true。

使用equals方法进行判断时则以具体equals方法中实际比较的项目为准，可通过重写equals方法实现自定义的比较方式

**public** **class** EqualsTest {

**public** **static** **void** main(String[] args) {

//基本数值类型的比较

**int** i = 65;

**float** d = 65.0f;

**char** s = 'A';

System.***out***.println(i==d); //true

System.***out***.println(i==s); //true

System.***out***.println(d==s); //true

//引用类型的比较

String str1 = **new** String("haha");

String str2 = **new** String("haha");

System.***out***.println(str1==str2); //false

System.***out***.println(str1.equals(str2)); //true

}

}

**public** **class** StringCompareTest {

**public** **static** **void** main(String[] args) {

String str1 = "guigu你好！";

String str2 ="guigu";

String str3 = "你好！";

String str4 = "guigu"+"你好！";

String str5 = "guigu"+"你好"+"！";

//上述字符串在编译的时候便可以确定字符串的内容，因此上述均是引用常量池的内容

String str6 = str2+str3;

//str6的内容在编译时无法直接确定，因此str6并不是引用常量池的内容，而是在运行时在堆内存中创建新的对象

String str7 = **new** String("guigu你好！");

//str7通过new创建新的对象，存在在堆内存中

System.***out***.println(str1==str4); //true

System.***out***.println(str1==str5); //true

System.***out***.println(str4==str5); //true

System.***out***.println(str1==str6); //false

System.***out***.println(str1==str7); //false

System.***out***.println(str1.equals(str4)); //true

System.***out***.println(str1.equals(str5)); //true

System.***out***.println(str4.equals(str5)); //true

System.***out***.println(str1.equals(str6)); //true

System.***out***.println(str1.equals(str7)); //true

}

}

### 重写equals方法

可通过重写equals方法实现按照指定的要求完成对两个对象的判断（比较）

（此处以Person类型为例，如果Person对象的id相同，可认为是同一个人，返回的是true）

**public** **class** Person {

**public** String name;

**public** **int** id;

**public** Person() { }

**public** Person(String name, **int** id) {

**super**();

**this**.name = name;

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

@Override

**public** **boolean** equals(Object obj) {

//判断是否指向同一个对象

**if**(**this**==obj)

**return** **true**;

//判断obj不为null，且两个被比较对象的类型相同

//（obj!=null&&obj instanceof Person）

**if**(obj!=**null**&&obj.getClass()==Person.**class**)

{

//如果obj是Person类型，则先将其进行强制类型转换

Person p = (Person)obj;

//如果两个对象的id相同则可认为两个比较对象相同，从而返回true

**if**(**this**.id==p.getId())

**return** **true**;

}

**return** **false**;

}

**public** **static** **void** main(String[] args) {

//测试

Person p1 = **new** Person("haha",1);

Person p2 = **new** Person("bibi",1);

Person p3 = **new** Person("haha",2);

System.***out***.println(p1.equals(p2)); //true

System.***out***.println(p1.equals(p3)); //false

}

}

5.方法的重载与方法的覆盖

* 方法的重载

定义：如果一个类中包含了两个以上的方法名相同，但是形式参数列表不同那么被称为方法的重载。

特点：“两同一不同”-->在同一个类中，方法名相同，形式参数列表不同(a.个数不同,b.位置不同.c 类型不同)，和返回值、修饰符无关。

* 方法的覆盖

定义：当子类继承父类并重写了父类的方法的这种行为被称为方法的覆盖。

特点：“两同两小一大“-->

两同即方法名相同，形式参数列表相同。

两小是指子类方法的返回值类型(协变类型(拥有继承关系))应该比父类更小，子类抛出的异常应该比父类更少更小。(继承关系、抛出异常)

一大是指子类的作用权限比父类的作用权限要大或者相等。

6.访问控制修饰符

Java提供了3个访问修饰符 private、 protected和public分别代表3个访问控制级别，除此以外系统还有一个不加任何访问修饰符的访问控制级别就是默认defalut，总共提供了4个访问修饰符。Java的访问控制修饰符控制级别由小到大如下:

private ------->default-------->protected------>public

访问控制修饰符的范围

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | private | default | protected | public |
| 同一个类中 | √ | √ | √ | √ |
| 同一个包中 |  | √ | √ | √ |
| 子类中 |  |  | √ | √ |
| 全局范围内 |  |  |  | √ |

Private : 私有的，只能本类中使用

Default : 默认的访问级别 可以在同一个类中，同一个包中使用

Protected ：受保护的级别 能在同一个类中，同一个包中，不同包的子类中使用

Public : 公有的，任意类都可以使用

7.单例模式

设计模式: 就是解决问题行之有效的方法。设计模式是一种思想。  
 单例模式：解决保证内存中始终只有一个对象。

如何保证内存中只有一个对象呢？

* 不让其他程序创建该类的对象
* 在本类中创建一个自己的对象
* 对外提供一个获取该对象的方法

实现的步骤：

* + 构造函数私有化
  + 提供一个静态的自己的私有的全局的变量
  + 然后提供一个get方法用来获取该全局变量

单例模式分为两种模式 懒汉式，饿汉式

懒汉式指的是在提供静态的自己的私有的全局变量的时候初始化为null，即后续相应的get方法返回该对象时需进行判断，如果为null则进行初始化

**public** **class** SingleTest1 {

/\*\*

\* 懒汉式

\*/

//1、将构造函数私有化

**private** SingleTest1()

{

}

//2、提供一个静态的自己的私有的全局的变量，并初始化为null

**private** **static** SingleTest1 *st* = **null**;

//3、通过相应的静态的get方法获取该对象

**public** **static** SingleTest1 getSingleTest()

{

**if**(*st=*=**null**)

*st* = **new** SingleTest1();

**return** *st*;

}

}

饿汉式指的是在提供静态的自己的私有的全局变量的时候直接创建对象，后续通过相应的get方法返回该对象

**public** **class** SingleTest2 {

/\*\*

\* 饿汉式

\*/

//1、构造函数私有化

**private** SingleTest2()

{

}

//2、定义一个静态的自己的私有的全局的变量,并通过构造函数初始化对象

**private** **static** SingleTest2 *st* = **new** SingleTest2();

//3、通过相应的静态的get方法直接返回该对象

**public** **static** SingleTest2 getSingleTest()

{

**return** *st*;

}

}

通过Go类进行测试，可看到如下结果

**public** **class** Go {

/\*\*

\* 通过Go类进行单例模式的测试

\*/

**public** **static** **void** main(String[] args) {

SingleTest1 st1 = SingleTest1.*getSingleTest*();

SingleTest1 st2 = SingleTest1.*getSingleTest*();

System.***out***.println(st1);

System.***out***.println(st2);

SingleTest2 st3 = SingleTest2.*getSingleTest*();

SingleTest2 st4 = SingleTest2.*getSingleTest*();

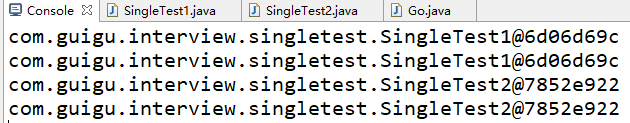
System.***out***.println(st3);

System.***out***.println(st4);

}

}

输出结果显示相应的都是同一个对象



8.引用类型的强制类型转换

基本数据类型发生类型转换分为自动类型转换和强制类型转换。引用类型的类型转换分为上溯造型 和向下转型

“上溯造型”是指 **子类声明父类接收**  把子类类型转换为父类

“向下转型 ”是指 **把父类的类型强制转换为子类**

如果抽象数据类型之间发生关系转换，必须有继承关系

举例分析：

**public** **class** Go {

//通过Go类测试相应的内容

**public** **static** **void** main(String[] args) {

//上溯造型：子类声明、父类接收

Person t1 = **new** Teacher();

t1.eat(); //显示的是子类重写后的信息

t1.walk();

//t1.teach();在编译（编写代码时）时t1的类型是Person类，此时Person类中并没有teach方法，因此出现错误

//但t1运行时的类型是Teacher类型

//向下转型：将父类类型强制转换为子类，即将Person强制转换为Teacher

Teacher t = (Teacher)t1;

t.teach();//此时编译通过，显示teach方法中的相关信息

//下述语句在编译时出现java.lang.ClassCastException类型转换错误异常

//不能够将Teacher强制转化为Student类型

Student s = (Student)t1;

s.study();

}

}

### Instanceof的使用

通过instanceof实现方法的通用性，利用上述的案例实现测试，使得无论传入的是父类类型还是相应的子类类型，都能够进行判断并执行该类对应的所有方法

Instanceof的作用是判断两个比较的对象类型是否相同，相同则返回true，否则返回false

**public** **class** InstanceOfTest {

//定义一个test方法，test方法的形参是父类类型（此处为Person类）,这个形参可以接收当前对象或者是当前对象的子类

//即test方法可以接收Person类的对象或者是Person的子类对象

**public** **void** test(Person p)

{

p.eat();

p.walk();

**if**(p **instanceof** Teacher)

{

//((Teacher) p).teach();

Teacher t = (Teacher)p;

t.teach();

}

**else** **if**(p **instanceof** Student)

{

//((Student) p).study();

Student s = (Student)p;

s.study();

}

}

**public** **static** **void** main(String[] args) {

//测试

InstanceOfTest iot = **new** InstanceOfTest();

iot.test(**new** Person());

iot.test(**new** Teacher());

iot.test(**new** Student());

}

}

9.抽象类与接口的对比

### 抽象类的相关概念

定义：抽象类和抽象方法都必须使用abstract修饰符进行定义，有抽象方法的类只能被定义为抽象类，抽象类中可以没有抽象方法。

抽象类和抽象方法的规则：

* 抽象类和抽象方法必须使用abstract修饰符进行修饰，抽象方法是没有方法体的。
* Public void f1(){} 抽象方法 public abstract void f2();
* 抽象类不能被实例化，无法通过new关键字来调用抽象类的构造函数创建实例。
* 抽象类可以包括成员变量，方法，构造器和初始化代码块以及内部类，这些定义的内容主要是被子类调用。
* 含有抽象方法的类一定是一个抽象类

作用：

抽象类是从多个具体类中抽象出来的父类，它具有相同特征的类中抽象出一个抽象类，以这个抽象类为模板，从而避免子类的随意设计。抽象类实现的就是一种模板策略，抽象类作为多个子类的通用模板，子类在抽象类的基础上进行扩展和改造

细节问题：

* abstract能修饰类 能修饰方法但是不能修饰变量
* 拥有抽象方法的类一定是一个抽象类
* 抽象类中可以拥有构造方法，而且构造方法还可以重载
* 抽象类不能和 final private static 共存
* Final 修饰的方法不能被重写 而abstract修饰的方法必须在子类中实现

举例：

//带有abstract关键字的类称之为抽象类

**public** **abstract** **class** Shape {

//1、抽象类中可定义变量（变量不能用abstract修饰）

**public** String color;

//2、抽象类中可以有构造方法，且构造方法还可进行重载

**public** **void** Shape(){ }

**public** **void** Shape(String Color)

{

**this**.color = color;

}

//3、抽象类中可以定义抽象方法，抽象方法不能有方法体(此处定义获取面积的抽象方法)

**public** **abstract** **void** getSize();

//4、抽象类中可以定义代码块

{

System.***out***.println("代码初始化...");

}

//5、抽象类中定义普通方法

**public** String getColor() {

**return** color;

}

**public** **void** setColor(String color) {

**this**.color = color;

}

//6、抽象类中可以定义内部类，以上内容均可供子类进行调用

}

**public** **class** Triangle **extends** Shape {

//子类继承抽象类，则必须实现抽象类中所有的抽象方法

**public** **int** a;

**public** **int** b;

**public** **int** c;

**public** **int** h;

**public** Triangle() {

}

**public** Triangle(String color,**int** a, **int** b, **int** c, **int** h) {

**super**.Shape(color);

**this**.a = a;

**this**.b = b;

**this**.c = c;

**this**.h = h;

}

@Override

**public** **void** getSize() {

System.***out***.println("面积："+(a\*h)/2);

}

}

**public** **class** Go {

**public** **static** **void** main(String[] args) {

//抽象类无法被实例化，即无法通过new关键字来调用抽象类的构造函数创建实例。 Shape s = new Shape();

Triangle t = **new** Triangle("红色",3,4,5,4);

t.getSize();

}

}

### 接口的相关概念

和类的定义不同 接口不再使用class而是使用interface，接口的语法规则如下:

[修饰符] interface 接口名 extends 父接口1

{

零个或者多个常量的定义

零个或者多个抽象方法的定义

零个或者多个内部类、接口

零个或者多个默认方法或类方法定义

}

修饰符可以是public 或者省略 ，如果省略采用默认的访问控制修饰符。

接口名只需要遵循标识符的规则即可

**接口可以继承接口 但是不能继承类**

接口是一种规范，所以接口中不能包含构造器和初始化代码块。

**接口中可以包含成员变量(只能是静态变量),方法(只能是抽象实例方法，类方法、默认方法)，内部类。**

接口中定义的静态变量，他们是与接口相关的，系统会自动为这些变量增加public、static和final修饰符，接口里的定义如下: “int MAX\_SIZE=30;”等价于“public static final int MAX\_SIXE =30；”

举例：

**public** **interface** Outer {

/\*\*1、接口中定义的成员变量只能是常量

\* 常量的定义规则为大写字母+下划线\_

\* 系统默认自动添加public、static、final修饰符

\* public final static MAX\_SIZE = 50;

\*/

**int** ***MAX\_SIZE*** = 50;

/\*\*2、接口中不能定义普通的方法

\* public void getSize()

\* {

\*

\* }

\*/

//3、接口中可以定义抽象方法，且抽象方法可以重载，其相应的方法修饰符为public或省略不写

**public** **void** out();

**public** **void** out(String msg);

//4、jdk1.8之后，新增了默认方法（即用default修饰的方法）

**default** **void** print(String ... msgs) //不定长参数

{

**for**(String s : msgs)

System.***out***.println(s);

}

**default** **void** test()

{

System.***out***.println("test...");

}

//5、在接口中定义类方法，需要用static修饰

**static** String staticTest()

{

**return** "接口中的默认的类方法";

}

}

**public** **class** OuterTest {

**public** **static** **void** main(String[] args) {

//接口是更为纯粹的抽象类，不能够被实例化（不能够通过new创建对象）

//Outer o = new Outer();

//可以通过接口名访问接口中的常量以及类方法

System.***out***.println(Outer.***MAX\_SIZE***);

System.***out***.println(Outer.*staticTest*());

}

}

接口的继承和类的继承不同，接口完全支持多继承，即一个接口可以拥有多个直接父接口，和类的继承相似，子类扩展了父类 将会获得父接口中定义的所有的抽象方法、常量。

举例：

1. 一个接口可以继承单个或多个接口

**interface** interfaceA

{

**int** ***PROP\_A*** = 10 ;

**abstract** **void** testA();

}

**interface** interfaceB

{

**int** ***PROP\_B*** = 20 ;

**abstract** **void** testB();

}

//一个接口可以继承单个或者多个接口

**interface** interfaceC **extends** interfaceA,interfaceB

{

**int** ***PROP\_C*** = 30 ;

**abstract** **void** testC();

}

**public** **class** InterfaceExtendsTest {

**public** **static** **void** main(String[] args) {

System.***out***.println(interfaceC.***PROP\_A***);

System.***out***.println(interfaceC.***PROP\_B***);

System.***out***.println(interfaceC.***PROP\_C***);

}

}

1. 一个类可以实现单个或多个接口

**interface** interfaceA

{

**int** ***PROP\_A*** = 10 ;

**abstract** **void** testA();

}

**interface** interfaceB

{

**int** ***PROP\_B*** = 20 ;

**abstract** **void** testB();

}

//一个类可以实现单个或者多个接口，且必须实现相应接口的所有抽象方法

**public** **class** InterfaceImplementsTest **implements** interfaceA,interfaceB {

@Override

**public** **void** testA() {

System.***out***.println("testA...");

}

@Override

**public** **void** testB() {

System.***out***.println("testB...");

}

**public** **static** **void** main(String[] args) {

InterfaceImplementsTest iit = **new** InterfaceImplementsTest();

iit.testA();

iit.testB();

System.***out***.println(iit.***PROP\_A***);

System.***out***.println(iit.***PROP\_B***);

}

}

### 两者之间的对比

相同之处：

* 接口和抽象类都不能被实例化，他们都是依赖于子类实现。
* 接口和抽象类都可以包含抽象方法，实现接口或者继承抽象类都必须实现这里定义的全部的抽象方法。

不同之处：

* 接口中只能包含抽象方法和默认方法，和类方法，不能为普通方法提供方法的实现。
* 接口中不可以定义构造函数、抽象类中可以定义构造函数
* 接口中不能定义初始化代码块，抽象类可以包含初始化代码块
* 一个类最多只能包含一个直接父类，但是接口可以有多个父接口。而且一个类可以实现多个接口、

10.匿名内部类与Lambda表达式

### 匿名内部类

匿名内部类是说只适合使用一次的类。匿名内部类不能被重复使用。

举例：

**interface** Product{

**public** **double** getPrice();

**public** String getName();

}

**public** **class** Computer **implements** Product {

**public** String getName() {

**return** "ACER";

}

**public** **double** getPrice() {

**return** 10;

}

}

**public** **class** AnonymousTest {

**public** **void** showProduct(Product product)

{

System.***out***.println("产品名称："+product.getName()+"价格:"+product.getPrice());

}

**public** **static** **void** main(String[] args) {

AnonymousTest at = **new** AnonymousTest();

//匿名内部类，Product是一个接口，需要通过子类实现接口中所有的抽象方法

//下述{ }就是匿名内部类，相当于前方接口Product的子类

at.showProduct(**new** Product(){

**public** String getName() {

**return** "宏基";

}

**public** **double** getPrice() {

**return** 30;

}

});

}

}

在jdk1.8之前要求被局部内部类、匿名内部类访问的局部变量必须是final修饰的，从jdk1.8之后取消了这个限制， 系统会自动添加一个final修饰

### Lambda表达式

Lambda表达式可以简化操创建匿名内部类的对象

* + 形式参数列表 是接口中需要的形参列表
  + 箭头(->)通过英文中划线和大于符号组成
  + 代码块 代码块就代表实现了这个接口并且可以重写方法。如果这个接口中只有一个方法 可以省略方法

案例分析：

**public** **interface** Command {

//在Command接口中定义一个抽象方法

**void** test(**int**[] target);

}

**public** **class** ProcessArray {

//封装接口Command中的test方法

**public** **void** process(**int**[] target,Command cmd)

{

cmd.test(target);

}

}

//使用匿名内部类实现接口

**public** **class** CommandTest {

**public** **static** **void** main(String[] args) {

//得到ProcessArray的一个对象

ProcessArray pa = **new** ProcessArray();

**int**[] array = {1,2,3,4,5};

//调用ProcessArray的方法process,传入相应的参数

//第一个参数为target整型数组，第二个参数是接口类型，则可使用匿名内部类实现该接口

pa.process(array, **new** Command(){

//实现Command接口中所有的抽象方法

**public** **void** test(**int**[] target)

{

**int** sum = 0;

**for**(**int** t : target)

{

sum += t;

}

System.***out***.println("sum:"+sum);

}

});

}

}

//使用Lambda表达式实现接口

**public** **class** CommandTest2 {

**public** **static** **void** main(String[] args) {

ProcessArray pa = **new** ProcessArray();

**int**[] array = {1,2,3,4,5};

/\*\*使用Lambda表达式实现Command接口

\* (int[] target)表示形式参数列表，即接口中需要的形式参数列表

\* 箭头：->

\* 代码块 { } ：实现接口并重写相应的方法，如果接口中只有一个方法，可以省略方法，直接实现方法体中的内容

\*/

pa.process(array, (**int**[] target)->{

**int** sum = 0;

**for**(**int** t : target)

{

sum += t;

}

System.***out***.println("sum:"+sum);

});

}

}

Lambda表达式和匿名内部类有如下相同点:

* Lambda表达式和匿名内部类一样 可以直接访问局部变量，以及外部类的成员变量。
* Lambda表达式创建的对象与匿名内部类生成的对象一致的。

//基本的Lambda表达式操作

**interface** Eateable

{

**void** taste();

}

**interface** Flyable

{

**void** fly(String weather);

}

**interface** Addable

{

**int** add(**int** a,**int** b);

}

**public** **class** LambdaTest1 {

**public** **void** eat(Eateable e)

{

System.***out***.println(e);

e.taste();

}

**public** **void** driver(Flyable f)

{

System.***out***.println("驾驶ing"+f);

f.fly("天气不错");

}

**public** **void** sum(Addable add)

{

System.***out***.println("1+1="+add.add(1, 1));

}

**public** **static** **void** main(String[] args) {

LambdaTest1 lt = **new** LambdaTest1();

//1、Lambda表达式的代码块只有一条语句,则可以省略{ }

lt.eat(()->{System.***out***.println("haha,很好吃");});

lt.eat(()-> System.***out***.println("haha,很好吃"));

//2、Lambda表达式只有一个形式参数，则可以省略（ ）

//如果有多个形式参数，则可以相应的省略类型声明，系统会自动匹配相应的数据类型

lt.driver((String weather)->{

System.***out***.println("今天"+weather);

System.***out***.println("aaaa");

});

lt.driver(weather->{

System.***out***.println("今天"+weather);

System.***out***.println("aaaa");

});

//3、Lambda表达式的代码块只有一条语句,则可以省略{ }

// 代码块中只有一条语句，需要返回值，亦可以省略return关键字

lt.sum((a,b)->{**return** a+b;});

lt.sum((a,b)-> a+b);

}

}

### 函数式接口

Lambda表达式是类型也被称为目标类型，Lambda表达式目标类型必须是函数式接口，所谓的函数式接口是指只有一个抽象方法的接口。函数式接口代表只包含一个抽象方法的接口，函数式接口可以包含多个默认方法，类方法，但只能声明一个抽象方法。

Java8专门为函数式接口提供了一个注解@FunctionalInterface，该注解通常放在定义接口的前方，该注解对程序没有任何作用，它仅仅是告诉编译器更加严格的检查当前接口是否是函数式接口。

Lambda表达式实现的是匿名内部类的替代，只能实现特定的函数式接口， 也就意味着Lambda表达式遵循以下两个限制

* **Lambda表达式目标类型必须是明确的函数式接口**
* **Lambda表达式只能为函数式接口创建对象**

为了保证Lambda表达式目标类型是一个明确的函数式接口，可以有三种常见的方式

* 将Lambda表达式赋值给函数式接口的变量
* 将Lambda表达式作为函数式接口的形参传递给某个方法
* 使用函数式接口对Lambda表达式进行强制类型转换

@FunctionalInterface

**interface** FkTest

{

**void** run();

}

**public** **class** LambdaTest2 {

**public** **static** **void** main(String[] args)

{

//Runnable是一个函数式接口，其只包含一个抽象方法，则可通过Lambda实现该将接口

Runnable r = ()->{ System.***out***.println("Runnable是一个函数式接口");};

//Object本身并不是一个函数式接口，无法通过Lambda表达式实现，

//但可以通过强制类型转化实现

Object obj = (Runnable)()->{

System.***out***.println("Object并不是一个函数式接口");

};

//Lambda表达式的形参列表与函数式接口中的抽象方法中的形参列表相同

Object obj2 = (FkTest)()->{

System.***out***.println("Object强制转化为自定义的函数式FkTest");

};

}

}

### 方法引用与构造器引用

如果labmda表达式代码块只有一条代码，程序就可以省略Lambda表达式中的{}

如果Lambda表达式只有一条代码，还可以在代码中使用方法引用和构造器引用。

Lambda表达式支持的方法引用和构造器引用

|  |  |  |  |
| --- | --- | --- | --- |
| 种类 | 示例 | 说明 | 对应的Lambda表达式 |
| 引用类方法 | 类名::类方法 | 函数式接口中被实现方法的全部参数传递给该类方法作为形参 | （a,b）->类名.类方法{a,b} |
| 引用特定对象的实例方法 | 特定对象::实例方法 | 函数式接口中被实现方法的全部参数传递给该类方法作为形参 | (a,b…)->特定对象.实例方法(a,b..) |
| 引用某类对象的实例方法 | 类名::实例方法 | 函数式接口中被实现方法的全部参数传递给该类方法作为形参 |  |
| 引用构造器 | 类名::new |  |  |

案例分析：

**import** javax.swing.JFrame;

@FunctionalInterface

//Converter为函数式接口

**interface** Converter

{

//形参为String类型，返回值类型为Integer

Integer convert(String from);

}

@FunctionalInterface

**interface** MyTest

{

String test(String a,**int** b,**int** c);

}

@FunctionalInterface

**interface** YourTest

{

JFrame win(String title);

}

**public** **class** MethodRefer {

**public** **static** **void** main(String[] args) {

//通过Lambda表达式创建Converter对象

Converter converter = from -> Integer.*valueOf*(from);

Integer i = converter.convert("123");

System.***out***.println(i);

//1、通过方法引用替代Lambda表达式,Conveter函数式接口中被实现的方法的

//全部参数“（String from）”传递给类方法

Converter converter1 = Integer::*valueOf*;

Integer i1 = converter.convert("123");

System.***out***.println(i1);

//2、方法引用代替Lambda表达式引用特定对象的实例方法

//得到字符串中指定字符的位置

System.***out***.println("helloguigu".indexOf("l"));

Converter converter2 = "helloguigu"::indexOf;

Integer i2 = converter2.convert("l");

//截取指定字符串，返回截取后的字符串内容

System.***out***.println("helloguigu".substring(3,6));

MyTest mt = (a,b,c) -> a.substring(b, c);

System.***out***.println(mt.test("helloguigu", 3, 6));

MyTest mt1 = String::substring;

System.***out***.println(mt1.test("helloguigu", 3, 6));

//3、引用构造器

YourTest yt = title -> **new** JFrame(title);

//相当于JFrame j = new JFrame(title)的效果

JFrame jf = yt.win("haha");

System.***out***.println(jf);

YourTest yt1 = JFrame::**new**;

JFrame jf1 = yt.win("haha");

System.***out***.println(jf1);

}

}

11.垃圾回收

常见的面试题：

自我学习，取了解垃圾回收的算法。至少掌握三种以上。

12.System类

**public** **class** ViewWork {

**public** ViewWork() {

System.***out***.println("1");

}

//err是带有缓冲输出的,并不是立即输出 ,而是当缓冲区满了,或者程序执行完毕才执行输出

{

System.***err***.println("2");

}

**static** {

System.***out***.println("3");

}

**public** **static** **void** main(String[] args) {

ViewWork vw =**new** ViewWork();

/\*\*在正常情况下在类加载时先执行static代码块，

\* 随后执行普通代码块，最后执行构造方法中的内容

\* 但由于err是带缓冲输出，因而输出的时间并不确定，

\* 从而导致每次输出的结果都不尽相同（err输出是红色显示）

\*/

}

}

13.精确运算与大数运算

### BigDecimal和BigInteger的运用

1. 如果对double类型的数据进行精确运算，则需要将其转化为BigDecimal类型

**import** java.math.BigDecimal;

**public** **class** BigDecimalTest1 {

**public** **static** **void** main(String[] args) {

//如果直接输出小数之间进行加减乘除运算的结果，可以看到输出结果有时候并不如我们所愿，这是由于不同的数据类型对数据的存储有所限制

/\*\*System.out.println(0.05+0.01); //0.060000000000000005

\* System.out.println(1.0-0.441); //0.5589999999999999

\* System.out.println(233.56/100); //2.3356

\* System.out.println(3.24\*100); //324.0

\*/

/\*\*BigDecimal提供不同的构造方法从而实现不同数据的转化

\* 亦提供了valueOf方法实现将long类型、double类型的数据转化为BigDecimal类型

\* 还提供了许多方法实现基本的数学运算(具体运算参考相应的api文档 java.math.BigDecimal)

\*/

//此处简单实现两个double类型的数据进行精确运算

BigDecimal b1 = BigDecimal.*valueOf*(0.05);

BigDecimal b2 = BigDecimal.*valueOf*(0.01);

BigDecimal sum = b1.add(b2);

BigDecimal sub = b1.subtract(b2);

BigDecimal mul = b1.multiply(b2);

BigDecimal div = b1.divide(b1);

System.***out***.println("和："+sum);

System.***out***.println("差："+sub);

System.***out***.println("乘积："+mul);

System.***out***.println("商："+div);

}

}

1. 设计一个大数的计算器

**import** java.math.BigDecimal;

**import** java.math.BigInteger;

**public** **class** BigDecimalTest2 {

**public** **static** **void** main(String[] args) {

//设计一个用于计算大数的计算器

//1、利用String类型存储要进行计算的数据

String str1 = "1111111111111111111111111111111.333";

String str2 = "2222222222222222222222222222222";

//2、将String类型的数据转化为相应的BigDecimal类型

//此处转化无法直接通过valueOf方法进行转化，因而借助BigDecimal的构造方法实现类型转换

/\*

BigDecimal b1 = new BigDecimal(str1);

BigDecimal b2 = new BigDecimal(str2);

BigDecimal sum = b1.add(b2);

BigDecimal sub = b1.subtract(b2);

BigDecimal mul = b1.multiply(b2);

BigDecimal div = b1.divide(b2);

System.out.println("和："+sum);

System.out.println("差："+sub);

System.out.println("乘积："+mul);

System.out.println("商："+div);

\*/

//BigInteger类型的使用和BigDecimal大同小异,但BigInteger的存储的数据类型需为整型，

//否则会抛出 java.lang.NumberFormatException异常信息

BigInteger b1 = **new** BigInteger(str1);

BigInteger b2 = **new** BigInteger(str2);

BigInteger sum = b1.add(b2);

BigInteger sub = b1.subtract(b2);

BigInteger mul = b1.multiply(b2);

BigInteger div = b1.divide(b2);

System.***out***.println("和："+sum);

System.***out***.println("差："+sub);

System.***out***.println("乘积："+mul);

System.***out***.println("商："+div);

}

}

### 简易计算器

**import** java.math.BigDecimal;

**public** **class** Calculator {

//设计简易计算器，默认计算精度

**private** **static** **final** **int** ***DEF\_DIV\_SCSLE*** = 10;

**public** **static** **double** add(**double** a,**double** b)

{

BigDecimal a1 = BigDecimal.*valueOf*(a);

BigDecimal b1 = BigDecimal.*valueOf*(b);

**return** a1.add(b1).doubleValue(); //将BigDecimal类型转化为Double类型

}

**public** **static** **double** sub(**double** a,**double** b)

{

BigDecimal a1 = BigDecimal.*valueOf*(a);

BigDecimal b1 = BigDecimal.*valueOf*(b);

**return** a1.subtract(b1).doubleValue();

}

**public** **static** **double** mul(**double** a,**double** b)

{

BigDecimal a1 = BigDecimal.*valueOf*(a);

BigDecimal b1 = BigDecimal.*valueOf*(b);

**return** a1.multiply(b1).doubleValue();

}

**public** **static** **double** div(**double** a,**double** b)

{

BigDecimal a1 = BigDecimal.*valueOf*(a);

BigDecimal b1 = BigDecimal.*valueOf*(b);

//第一个参数是被除数 第二个参数是保留的位数 第三个参数是进位

**return** a1.divide(b1, ***DEF\_DIV\_SCSLE***, BigDecimal.***ROUND\_HALF\_UP***).doubleValue();

}

**public** **static** **void** main(String[] args) {

System.***out***.println("0.05+0.03的值为"+Calculator.*add*(0.05, 0.03));

}

}

14.日期相关

### Date类

**import** java.util.Date;

**public** **class** DateTest {

**public** **static** **void** main(String[] args) {

//创建date对象，打印当前系统时间

Date d = **new** Date();

System.***out***.println(d);

//得到当前时间之后1000ms的时间（延迟1000ms）

Date d1 = **new** Date(System.*currentTimeMillis*()+1000);

System.***out***.println(d1);

//比较d和d1对象

System.***out***.println(d.before(d1)); //true

System.***out***.println(d1.after(d)); //true

System.***out***.println(d.compareTo(d1)); //-1

//获取某个日期相应的年、月、日

//由于Date版本的某些方法失效，对于获取年份要在其基础上加上1900，获取月份则要在其基础上加上1

//其余获取日期、星期则保持不变

System.***out***.println("年："+d.~~getYear~~()+1900);

System.***out***.println("月："+d.~~getMonth~~()+1);

System.***out***.println("日："+d.~~getDate~~());

System.***out***.println("星期"+d.~~getDay~~());

}

}

### Calendar类

**案例1：**

**import** java.util.Calendar;

**import** **static** java.util.Calendar.\*;

**public** **class** CalendarTest {

**public** **static** **void** main(String[] args) {

//获取Calendar的一个实例

Calendar cal = Calendar.*getInstance*();

System.***out***.println(cal);

//依次直接得到相应的年月日、星期

//此处要导包“import static java.util.Calendar.\*;”，将相应的静态变量导入

//月份要执行加1操作，其余内容不作改变

System.***out***.println("年："+cal.get(***YEAR***));

System.***out***.println("月："+(cal.get(***MONTH***)+1));

System.***out***.println("日："+cal.get(***DATE***));

//输出其它相关的信息

//设置一个指定的日期

cal.set(2017, 10, 18, 14, 22, 23); //设置的时间为2017-11-18 14:22:23

System.***out***.println(cal.getTime());

//得到下述信息：今天是这周的第几天 今天是这一月的第几天 今天是这一年的第几天

System.***out***.println(cal.get(***DAY\_OF\_WEEK***));

System.***out***.println(cal.get(***DAY\_OF\_MONTH***));

System.***out***.println(cal.get(***DAY\_OF\_YEAR***));

//计算某个同学出生那天是周几

//1998-03-11

cal.set(1998, 2, 11);

//这周的第几天与这周的星期几有所区别，外国人的星期是从周日开始计算，周日为一周的第一天

System.***out***.println("这个同学出生那天是周:"+(cal.get(***DAY\_OF\_WEEK***)-1));

}

}

**案例2：**

**import** java.util.Calendar;

**public** **class** CalendarTest2 {

**public** **static** **void** main(String[] args) {

Calendar cal = Calendar.*getInstance*();

//另一种方式：设置指定日期

cal.set(Calendar.***YEAR***, 1998);

cal.set(Calendar.***MONTH***, 02);

cal.set(Calendar.***DATE***, 11);

System.***out***.println(cal.getTime());

//计算到目前为止活了几天(当前毫秒数减去出生日期的毫秒数，再转化为相应的天数)

System.***out***.println((System.*currentTimeMillis*()-cal.getTimeInMillis())/1000/60/60/24);

}

}

面试题：有以下代码，尝试分析输出的结果为？

**public** **static** **void** main(String[] args) {

Calendar cal = Calendar.*getInstance*();

cal.set(Calendar.***YEAR***, 2016);

cal.set(Calendar.***MONTH***, 1);

cal.set(Calendar.***DATE***, 32);

System.***out***.println(cal.getTime());

}

分析：2016年为闰年，此处的月份本应为1+1=2月，但由于此时日期为32，则有闰年的2月份有29天，则说明32-29=3，正确解析应为2016年3月3号

### 位符的格式化

SimpleDateFormat格式

**import** java.text.SimpleDateFormat;

**import** java.util.Date;

**public** **class** SimpleDateFormatTest {

**public** **static** **void** main(String[] args) {

Date d= **new** Date();

//格式 其他直接输出

//G 公元 yyyy年 D指定日期是这一年的第几天

SimpleDateFormat sdf =**new** SimpleDateFormat("Gyyyy年中的第D天");

String str=sdf.format(d);

System.***out***.println(str);

SimpleDateFormat sdf2 =**new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

System.***out***.println(sdf2.format(d));

}

}

15.正则表达式

### 正则表达式的创建

|  |  |
| --- | --- |
| *x* | 字符 *x* |
| \\ | 反斜线字符 |
| \0*n* | 带有八进制值 0 的字符 *n* (0 <= *n* <= 7) |
| \0*nn* | 带有八进制值 0 的字符 *nn* (0 <= *n* <= 7) |
| \0*mnn* | 带有八进制值 0 的字符 *mnn*（0 <= *m* <= 3、0 <= *n* <= 7） |
| \x*hh* | 带有十六进制值 0x 的字符 *hh* |
| \u*hhhh* | 带有十六进制值 0x 的字符 *hhhh* |
| \t | 制表符 ('\u0009') |
| \n | 新行（换行）符 ('\u000A') |
| \r | 回车符 ('\u000D') |
| \f | 换页符 ('\u000C') |
| \a | 报警 (bell) 符 ('\u0007') |
| \e | 转义符 ('\u001B') |
| \c*x* | 对应于 *x* 的控制符 |

|  |  |
| --- | --- |
| [abc] | a、b 或 c（简单类） |
| [^abc] | 任何字符，除了 a、b 或 c（否定） |
| [a-zA-Z] | a 到 z 或 A 到 Z，两头的字母包括在内（范围） |
| [a-d[m-p]] | a 到 d 或 m 到 p：[a-dm-p]（并集） |
| [a-z&&[def]] | d、e 或 f（交集） |
| [a-z&&[^bc]] | a 到 z，除了 b 和 c：[ad-z]（减去） |
| [a-z&&[^m-p]] | a 到 z，而非 m 到 p：[a-lq-z]（减去） |

|  |  |
| --- | --- |
| **预定义字符类** | |
| . | 任何字符（与行结束符可能匹配也可能不匹配） |
| \d | 数字：[0-9] |
| \D | 非数字： [^0-9] |
| \s | 空白字符：[ \t\n\x0B\f\r] |
| \S | 非空白字符：[^\s] |
| \w | 单词字符：[a-zA-Z\_0-9] |
| \W | 非单词字符：[^\w] |

|  |  |
| --- | --- |
| **Greedy 数量词** | |
| *X*? | *X*，一次或一次也没有 |
| *X*\* | *X*，零次或多次 |
| *X*+ | *X*，一次或多次 |
| *X*{*n*} | *X*，恰好 *n* 次 |
| *X*{*n*,} | *X*，至少 *n* 次 |
| *X*{*n*,*m*} | *X*，至少 *n* 次，但是不超过 *m* 次 |

案例分析：

**public** **class** RegDemo1 {

**public** **static** **boolean** check(String qq)

{

//验证QQ号的合法性 1、不能0开头 2、 长度 5-15位 3、 必须都是数字

**boolean** check = **false**;

//通过嵌套的if...else...语句实现

**if**(qq.length()>=5&&qq.length()<=15)

{

**if**(!qq.startsWith("0"))

{ //利用异常处理非法字符的信息，亦可将相应的字符串转化为字符数组一一进行判断

**try** {

**long** l = Long.*parseLong*(qq);

check = **true**;

} **catch** (NumberFormatException e) {

System.***out***.println("QQ中存在非法字符..");

}

}

**else**

System.***out***.println("qq账号不能以0为开头");

}

**else**

System.***out***.println("字符串长度必须为5-15位之间");

**return** check;

}

**public** **static** **void** main(String[] args) {

String qq = "123456";

System.***out***.println(RegDemo1.*check*(qq));

//利用正则表达式实现数据的验证（验证qq账号的合法性，验证邮箱账号的合法性）

//可通过正则表达式直接替换下述的内容（check方法）

//1、先定义一个正则表达式

String reg = "[1-9][0-9]{4,14}";

//2、通过match方法进行判断，看相应的字符串是否符合相应的内容

System.***out***.println(qq.matches(reg));

}

}

### 正则表达式的应用

案例分析：

**import** java.util.regex.Matcher;

**import** java.util.regex.Pattern;

**public** **class** RegDemo2 {

/\*\*

\* 正则表达式 是对字符串的操作

\* 1.匹配 --> 使用字符串的mathes（String reg ）方法

\*

\* 2.切割 --> 使用String类中的split(String reg) 这个方法

\*

\* 3.替换 --> 使用String类中的replaceAll()方法

\*

\* 4.获取 --> 由于String没有提供直接的方法 获取 内容 aaaaaaab b

\* Pattern p =Pattern.compile("a\*b");

\* Mather m=p.matcher("aaab");

\* boolean b =m.matches();

\*/

//1、匹配：检验手机号码的合法性

**public** **static** **boolean** check()

{

String tel = "18816775859";

//1、定义正则表达式，用于检验手机号码的合法性

String reg = "1[34578]\\d{9}";

//此处手机号码以1开头，第二个数字为34578中的一个,\d需要通过反斜杠\进行转义，{9}表示前面的数据可以取9个

//通过字符串的matches方法进行判断

**return** tel.matches(reg);

}

//2、切割：根据指定的正则表达式的规则对字符串进行切割

/\*\*split方法切割字符串形式比较单一，没办法很好的解决一些特殊字符串以及特殊形式的切割规则的切割问题，

\* 对于特殊字符串的切割可以考虑StringTokenizer类进行解决：

\* StringTokenizer st = new StringTokenizer(str, "特殊字符串");

\* 对于重复字符的切割，由于没办法直接确定分割字符的个数，因此考虑借助正则表达式实现

\*/

**public** **static** **void** splitDemo()

{

String str = "34 54 12 67 89 1";

//1、定义相应的正则表达式

String reg = " +"; //由一个及以上的“ ”（空格）进行分割

//2、利用字符串的split方法实现字符串的分割，并打印相关的信息

**for**(String s : str.split(reg))

{

System.***out***.print(s+"-");

}

System.***out***.println();

//如果要求是对于下方字符串出现两次以上相同的字符作为切割位置(步骤类似进行设计)

String str1 = "aabcdddeffg";

// .代表任意 字符 (.)代表一个任意字符 而且是一个组

// \\1 代表取前方的一组数据 +代表 \\1可以出现至少一次 或多次

String reg1 = "(.)\\1+"; // 即整个表达式是两次以上的叠词

**for**(String s : str1.split(reg1))

{

System.***out***.print(s+"-");

}

}

//3、替换：根据指定的正则表达式的规则替换字符串中的部分内容

**public** **static** String replace()

{

//叠字处理：对于连续出现两次以上的字符，将其替换为指定的内容或者是只保留一个即可

String str = "abbbccdefffgg";

String str1 = str.replaceAll("(.)\\1+", "@");

// String str1 = str.replaceAll("(.)\\1+", "$1");

**return** str1;

/\*\*

//打码操作：对手机内容指定位置部分进行打码

//1、定义要进行替换的字符串

String str = "15868154584";

//2、借助字符串的replaceAll方法实现整体替换（对手机号码的第4-7位进行打码）

//对于每一组数据要用小括号（）进行标识，否则无法识别

//$1 指的是取前面的第一组数据，依次类推$n指的是取前面的第n组数据

str = str.replaceAll("(\\d{3})(\\d{4})(\\d{4})","$1\*\*\*\*$3" );

return str;

\*/

}

//4、获取：获取指定内容的字符串

**public** **static** **void** getString()

{

//1、定义进行测试的数据

String str = "aaa bb c dd eee gggg ffff ada xixi lala gog";

//2、定义相应的正则表达式，筛选出在上述的字符串中出现具有三个字符的内容

// String reg = "([a-z])\\1{3}";//相同的数据连续出现四次，错误的分析

String reg ="\\b[a-z]{3}\\b"; //?????????? \\b起着什么作用？

Pattern p =Pattern.*compile*(reg);

Matcher m =p.matcher(str);

//遍历查询数据即可

**while**(m.find()) {

System.***out***.println(m.start()+"----"+m.group()+"-----"+m.end());

}

}

**public** **static** **void** main(String[] args) {

// System.out.println(RegDemo2.check());

// RegDemo2.splitDemo();

// System.out.println(RegDemo2.replace());

RegDemo2.*getString*();

}

}

实战练习：

**import** java.util.Arrays;

**public** **class** RegTest{

/\*\*

\* 1、 我.我我..我我我我...我我我..我我我我我..我我我..我要要要..要要要..要要要..

\* 要要..要要.要.要要学..学学..学..学学..学学学..学学..学学..学学学编编.编编..

\* 编.编编..编编.编编编.程程程.程程.程程.程.程程..程程;

\* 还原为 "我要学编程"

\*

\* 2、 通过网路爬虫获取了一段ip地址 要求对这个ip地址排序

\* String ip="10.10.10.10 192.168.1.112 192.168.3.2

\* 192.168.1 111 202.111.1.1 3.3.3.3

\* 128.1.1.1 193.165.14.12";

\* 对上方ip地址进行排序

\*

\* 3、校验邮箱的合法性 正则表达式校验

\*/

//1、还原:去除字符“.”、叠词删减为相应的一个

**public** **static** **void** f1()

{

String str = "我.我我..我我我我...我我我..我我我我我..我我我..我要要要..要要要.."

+ "要要要..要要..要要.要.要要学..学学..学..学学..学学学..学学..学学..学学学"

+ "编编.编编..编.编编..编编.编编编.程程程.程程.程程.程.程程..程程";

//.表示任意字符，要通过转义字符\将其进行转化为普通的小点

str = str.replaceAll("\\.+", "");

str = str.replaceAll("(.)\\1+", "$1");

System.***out***.println(str);

}

//2、比较随机ip地址的大小

**public** **static** **void** f2() {

/\*\*

\* 排序 1.统一所有的位数，都保留三位：a)所有的数据前方都补两个0 b)从后方截取字符串保留三位即可

\*/

String ip = "10.10.10.10 192.168.1.112 192.168.3.2 192.168.1.111 202.111.1.1 3.3.3.3 128.1.1.1 193.165.14.12";

// 所有的数字前方添加两个0

ip = ip.replaceAll("(\\d+)", "00$1");

// 截取后三位的数字

ip = ip.replaceAll("0\*(\\d{3})", "$1");

// 按照空格切分字符串 分为单独的一个ip地址

String[] ips = ip.split(" +");

Arrays.*sort*(ips);

//按照原有的格式打印排序后的ip顺序

**for** (String ipSort : ips) {

System.***out***.println(ipSort.replaceAll("0\*(\\d+)", "$1"));

}

// System.out.println(ip);

}

//3、校验邮箱的合法性

**public** **static** **void** f3() {

//定义存储邮箱信息的字符串

String email = "ab\_cdA12345@sina.com.cn";

//定义正则表达式实现邮箱校验

String reg = "\\w+@[a-zA-Z0-9]+(\\.[a-zA-Z]{2,}){1,3}";

//通过字符串的matches方法进行匹配

**boolean** b = email.matches(reg);

System.***out***.println(b);

}

**public** **static** **void** main(String[] args) {

RegTest.*f1*();

RegTest.*f2*();

RegTest.*f3*();

}

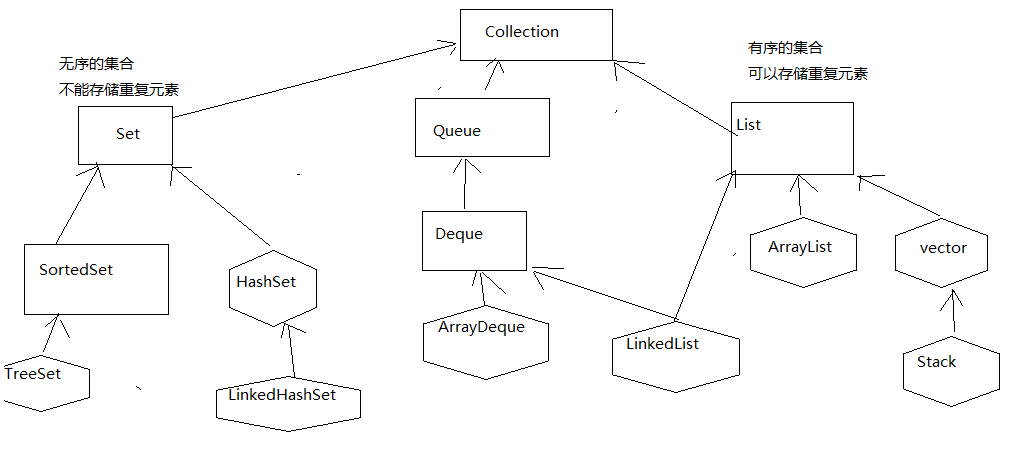
}

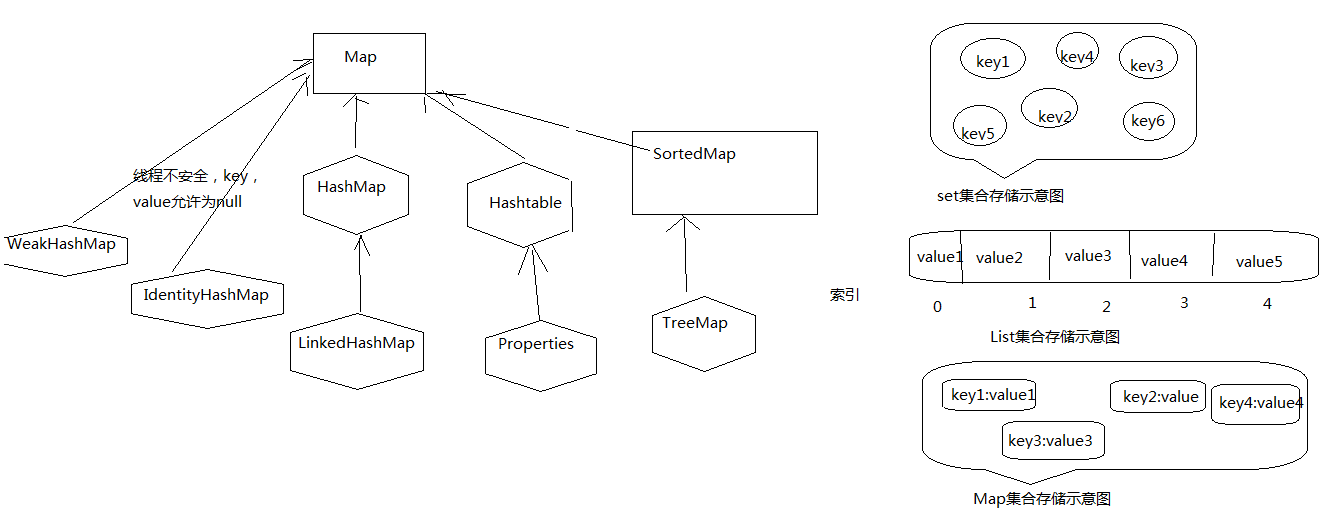
16.集合

为了保存数量不确定的数据以及保存具有映射关系的数据，Java提供了集合类，集合类主要是负责保存，存储其他数据，所以集合类也被称为容器类。所有的集合类都位于java.util包下，后来为了处理多线程并发的安全问题，java在jdk1.5之后提供了java.util.concurrent包提供了一些支持多线程的集合类。

集合类和数组不同，数组元素即可以是基本类型的数值，也可以是对象，而集合里只能保存对象，集合里保存的是基类型的包装类即对象。

Java的集合主要由两个接口派生而来，Collection和Map，这两个接口是根接口，这两个接口又包含了子类和实现类。





Collection中Set、Queue、List等集合

List集合是有序的集合，能够存储重复元素

* ArrayList和Vector作为List的两个实现类，功能完全相同，可以相互替代。Vector是一个线程安全的类，而ArrayList是线程不安全的类。
* 什么时候都不推荐使用Vector。可以通过Collections工具类把ArrayList转换为线程安全的类。
* 各种线性表性能的分析
* Java提供了一个线性接口。而ArrayList/Vector/LinkedList都是其子类。
* ArrayList和Vector底层都是数组结构，而LinkedList集合底层是链表结构。
* Vector是线程安全的，其他两个不是线程安全的。
* 如果需要经常执行插入、删除操作改变含有大量数据的List集合的大小 可以使用LinkedList。使用ArrayList和Vector会重新分配内部数组的大小和位置，所以性能相当于LinkedList较低。
* 如果经常用到查找和修改操作 可以使用ArrayList和Vector。

Set集合是无序的集合，能够存储不重复的元素

* HashSet是set接口的典型实现，HashSet是按照Hash算法来存存储集合中的数据，所以具有很好的存储和查询的性能，HashSet具有以下特点：
* 它不保证 set 的迭代顺序；特别是它不保证该顺序恒久不变
* HashSet不是同步的(不是线程安全的),如果多个线程同时访问一个HashSet，必须通过其他方式保证其同步。
* 集合中元素的值可以为null，但是只能有一个为null
* HashSet还有一个子类LinkedHashSet, LinkedHashSet集合是根据元素的hashCode值决定元素的存储位置，通过链表维护元素次序。
* TreeSet基于 TreeMap 的 NavigableSet 实现。使用元素的自然顺序对元素进行排序，或者根据创建 set 时提供的 Comparator 进行排序，具体取决于使用的构造方法。
* TreeSet自然排序：如果是基本数值类型 ，系统会自动实现排序，如果是抽象数据类型要实现比较必须实现Comparable接口，然后指定排序规则。
* 自定义排序：自定义排序是根据Treeset的构造函数进行指定，TreeSet的构造函数可以接受一个comparator的函数式接口，所有可以使用Lambda表达式进行相关的排序操作。

Map用于保存具有映射关系的数据，所以map集合里保存的是一组数值，Map里以key-value的形式保存数据，其中key和value都可以是引用类型。其中Map里的key不允许出现重复，如果出现重复将会发生覆盖。

* HashMap不保证迭代顺序，举例分析：

**public** **static** **void** main(String[] args) {

Map<String, String> map = **new** HashMap<>();

map.put(String.*valueOf*(System.*currentTimeMillis*()) + "a", "1");

map.put(String.*valueOf*(System.*currentTimeMillis*()) + "b", "2");

map.put(String.*valueOf*(System.*currentTimeMillis*()) + "c", "3");

map.put(String.*valueOf*(System.*currentTimeMillis*()) + "d", "4");

// 顺序是不确定的 是由于 HashMap不是按照固定位置存储 所以不保证迭代顺序

// 输出结果可能为1234/2314等......不同顺序

map.forEach((key, value) -> System.***out***.println(value));

}

* 分析HashMap底层源码的实现
* HashMap和Hashtable都是map集合的子类 并且他们之间的关系和ArrayList和Vector之间的关系是相同的，两种功能基本类似，并且在没有并发的情况下可以相关替代。
* Hashtable是一个线程安全的类，HashMap是线程不安全的类，所以HashMap性能比Hashtable要好一些，但是如果有多个线程访问Map使用Hashtable类会更好一些。
* Hastable不允许使用Null作为key和value。而HashMap允许使用null作为key和value值。HashMap中也只能有一个key值为null,而value不限制
* HashSet有一个子类LinkedHashSet，同理 HashMap下也有一个子类，这个子类是LinkedHashMap,使用双向链表维护key-value值的次序，该链表负责维护map迭代的顺序和插入的顺序是一致的。
* TreeMap是根据key值进行对map集合排序
* WeakHashMap：以弱键实现的基于哈希表的Map。在WeakHashMap 中，当某个键不再正常使用时，将自动移除其条目。更精确地说，对于一个给定的键，其映射的存在并不阻止垃圾回收器对该键的丢弃，这就使该键成为可终止的，被终止，然后被回收。丢弃某个键时，其条目从映射中有效地移除，因此，该类的行为与其他的 Map 实现有所不同。
* IdentityHashMap：IdentityHashMap 的key值是根据对象判断是否是同一个对象，如果不是同一个对象，key值key保存相同的数据

### 集合的几种迭代方式

无论是哪种迭代方式，在迭代的过程中均不能够对集合的大小作出修改（即不能直接在迭代过程中删除某个集合数据），否则报错

1、使用foreach语句遍历集合元素

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** ForeachTest {

**public** **static** **void** main(String[] args) {

//通过foreach语句迭代集合

Collection c = **new** HashSet();

c.add("语文");

c.add("数学");

c.add("英语");

**for**(Object o : c)

{

System.***out***.println(o);

}

}

}

2、使用lambda表达式遍历集合

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** LambdaTest {

**public** **static** **void** main(String[] args) {

//使用Lambda表达式遍历集合

Collection c = **new** HashSet();

c.add("语文");

c.add("数学");

c.add("英语");

//使用jdk8新增的forEach迭代集合

//c.forEach(Consumer action); Consumer是一个函数式接口，可以通过lambda表达式实现

c.forEach((obj)->{

System.***out***.println(obj);

});

}

}

3、使用Java8增强的Iterator遍历集合元素

**import** java.util.Collection;

**import** java.util.HashSet;

**import** java.util.Iterator;

**public** **class** IteratorTest {

**public** **static** **void** main(String[] args) {

Collection c = **new** HashSet();

c.add("语文");

c.add("数学");

c.add("英语");

//1、通过iterator接口简单实现集合的迭代

//把Collection集合转换为Iterator迭代器

Iterator iter = c.iterator();

//遍历迭代器中的每一个元素

**while**(iter.hasNext())

{

System.***out***.println(iter.next());

}

//2、通过java8增强的iterator，通过使用lambda表达式表示集合

//使用增强的方法forEachRemaining Consumer是一个函数式接口

iter.forEachRemaining((obj)->{

System.***out***.println(obj);

});

}

}

4、使用Java8新增的Predicate

通过传递的Predicate条件过滤集合

案例分析：

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** PredicateTest {

**public** **static** **void** main(String[] args) {

Collection c = **new** HashSet();

c.add("语文");

c.add("数学");

c.add("英语");

c.add("数据分析与应用");

//使用Lambda表达式完成条件的过滤 :c.removeIf(Predicate filter)

//Predicate是一个函数式接口:obj指传入每个迭代的参数，箭头之后的内容用以表示具体的过滤条件

c.removeIf((obj)-> ((String)obj).length()>5 );

System.***out***.println(c);

}

}

通过自定义的方法完成条件的过滤

案例分析：

**import** java.util.Collection;

**import** java.util.HashSet;

**import** java.util.function.Predicate;

**public** **class** PredicateTest2 {

//通过自定义的方法完成条件的过滤

/\*\*

\* 根据传递的Predicate条件 统计 books中满足条件的内容

\*

\* **@param** books

\* **@param** p

\* **@return**

\*/

**public** **static** **int** sumAll(Collection books, Predicate p) {

**int** total = 0;

// 循环集合中的数据,然后把集合中每个数据和条件进行匹配

**for** (Object obj : books) {

// 拿Predicate中的条件进行匹配,匹配成功则返回true，打印匹配的数据信息并进行统计

**if** (p.test(obj)) {

System.***out***.println(obj);

total++;

}

}

**return** total;

}

**public** **static** **void** main(String[] args) {

Collection c = **new** HashSet();

c.add("语文");

c.add("数学");

c.add("英语");

c.add("离散数学");

c.add("数据分析与应用");

//Predicate是函数式接口，通过lambda表达式实现

//1、筛选带有“数学”子串的数据的数量

System.***out***.println(PredicateTest2.*sumAll*(c, (obj)-> ((String)obj).contains("数学")));

//2、统计书名长度大于5的数据的数量

System.***out***.println(PredicateTest2.*sumAll*(c, (obj)-> ((String)obj).length()>5));

}

}

5、使用Java8新增的Stream

Java8新增了Stream，IntStream,LongStream,DoubleStream等流式的API这些API代表多个支持串行和并行聚集操作的元素。

独立使用Stream的步骤：

* 使用Stream的XxxStream的bulider方法创建Stream对应的Bulider对象
* 重复调用Bulider对象的add方法添加元素
* 调用Bulider对象的bulider方法 获取Stream
* 可以使用Stream中的聚集方法

案例1分析：

**import** java.util.Collection;

**import** java.util.HashSet;

**import** java.util.stream.IntStream;

**import** java.util.stream.IntStream.Builder;

**public** **class** StreamTest {

**public** **static** **void** main(String[] args) {

//IntStream is = IntStream.builder().add(20).add(30).add(40).add(-5).build();

Builder builder = IntStream.*builder*();

builder.add(20);

builder.add(30);

builder.add(40);

builder.add(-5);

IntStream is = builder.build();

//调用相应的聚集方法，但调用聚集方法的代码只能执行一次,连续执行出错

//1、求最大值max()、最小值min()

// System.out.println(is.max().getAsInt());

// System.out.println(is.min().getAsInt());

//2、求和sum()（求所有数据的总和）

// System.out.println(is.sum());

//3、统计数据的个数count()

// System.out.println(is.count());

//4、求解平均值average()

// System.out.println(is.average().getAsDouble());

//判断该Stream中是否所有的元素均大于或是小于某个值 is.allMatch(Preaicate predicate)

// System.out.println("is中所有的元素均大于10:"+is.allMatch( obj -> obj>10));

// System.out.println("is中所有的元素均大于-10:"+is.allMatch( obj -> obj>-10));

//迭代该Stream(方法引用：类名：：实例方法)

// is.forEach(System.out::println);

// is.forEach(obj->System.out.println(obj));

}

}

案例2分析：

**import** java.util.Collection;

**import** java.util.HashSet;

**import** java.util.stream.IntStream;

**public** **class** CollectionStream {

**public** **static** **void** main(String[] args) {

Collection c = **new** HashSet();

c.add("语文");

c.add("数学");

c.add("英语");

c.add("离散数学");

c.add("数据分析与应用");

//把Collection集合通过stream方法转换为Stream对象，从而可以使用Stream中的聚集方法以及其他统计的方法

//1、筛选带有“数学”子串的数据的数量

System.***out***.println(c.stream().filter(obj->((String)obj).contains("数学")));

//2、统计书名长度大于5的数据的数量

System.***out***.println(c.stream().filter(obj->((String)obj).length()>5));

//把Collection集合转换Stream对象 然后再调用Stream对象的mapToInt方法获取原有Stream对应的IntStream

System.***out***.println("-----");

IntStream is=c.stream().mapToInt(ele->((String)ele).length());

is.forEach(System.***out***::println);

}

}

### 工具类Collections和Arrays

Java提供了一个操作set list map 等集合的工具类，Collection工具类提供了大量的方法 对集合完成排序，查询，修改等操作。

17.泛型

Java集合有一个缺点，把一个对象丢进集合中，集合就会忘记这个对象的数据类型，该对象的类型就变为Object类型。Java集合之所以这样设计 因为集合的设计者不知道要存储的类型是什么，为了做到通用性设计为Obejct 。

为了解决这个问题 设计了泛型的概念。所谓的泛型就是在创建集合的时候指定集合的类型。

举例分析：

**import** java.util.ArrayList;

**import** java.util.HashSet;

**import** java.util.TreeMap;

**public** **class** ArrayListDemo {

**public** **static** **void** main(String[] args) {

//在编译的时候不检查类型的异常

ArrayList al = **new** ArrayList();

al.add(1);

al.add("haha");

al.add('k');

al.add(1.23333);

//上述集合添加数据操作，编译器正常执行，不显示任何错误操作，不检查错误

//在创建集合的时候指定集合存储数据的类型称之为泛型

ArrayList<String> als = **new** ArrayList<String>();

als.add("haha");

als.add("xixi");

//als.add(1); 一旦指定了集合数据存储的数据类型，就必须添加指定类型的数据，否则编译器会报错

//java7的菱形语法，后方的数据类型可以省略，系统会自动判断数据类型

ArrayList<Integer> ali = **new** ArrayList<>();

HashSet<Double> hs = **new** HashSet<>();

TreeMap<String,String> tm = **new** TreeMap<>();

}

}

### 深入泛型

案例分析：

//定义泛型类和泛型接口

**public** **class** Apple<T> {

//使用T类型的参数定义实例变量

**private** T info;

**public** Apple(){ };

**public** Apple(T info){

**this**.info = info ;

}

**public** T getInfo()

{

**return** **this**.info;

}

**public** **void** setInfo(T info)

{

**this**.info = info ;

}

}

//子类继承泛型父类 需要明确数据类型 或者 继续使用泛型

**public** **class** DemoA<String> **extends** Apple {

@Override

**public** String getInfo() {

**return** (String)**super**.getInfo();

}

}

**public** **class** DemoB<T> **extends** Apple {

@Override

**public** Object getInfo() {

**return** **super**.getInfo();

}

}

### 使用泛型通配符

**案例分析：**

**import** java.util.ArrayList;

//为了表示各种泛型List的父类 可以使用类型通配符。类型通配符是一个问号(?)。

//将一个问号作为类型实参传递给List集合。

**public** **class** GenDemo {

**public** **void** f1(ArrayList<Object> list)

{

//对list集合进行迭代

list.forEach(System.***out***::println);

}

**public** **void** f2(ArrayList<String> list)

{

list.forEach(System.***out***::println);

}

**public** **void** f3(ArrayList<?> list)

{

list.forEach(System.***out***::println);

}

**public** **static** **void** main(String[] args) {

GenDemo gd = **new** GenDemo();

ArrayList<String> list = **new** ArrayList<>();

list.add("123");

list.add("234");

list.add("345");

//gd.f1(list); ArrayList<Object> 不是ArrayList<String>的父类,因此编译不通过

gd.f2(list); //编译通过

gd.f3(list); //ArrayList<?>是ArrayList<String>的父类，编译通过

}

}

**设置通配符的上限和下限**

当直接使用List<?>这种形式，表明List集合可以是任何泛型List的父类 ,但是有一种特殊的情况，程序不希望这个List<?>是任何泛型List的父类 而是希望是某一种泛型List 的父类。

案例分析：

**import** java.util.ArrayList;

**import** java.util.Collection;

//定义Person、Worker、Teacher、Student四个类进行测试

**public** **class** GenDemo1 {

**public** **static** **void** show(Collection<?> list )

{

list.forEach(System.***out***::println);

}

**public** **static** **void** main(String[] args) {

/\*\*

\* 泛型的限定

\* ? extends E: 接受E类型 或者E的子类类型

\* ? super E: 接受E类型 或者E的父类类型

\*/

ArrayList<Person> ap =**new** ArrayList<>();

ap.add(**new** Person("张三1", 11));

ap.add(**new** Person("张三2", 12));

ap.add(**new** Person("张三3", 13));

ap.add(**new** Person("张三4", 14));

ArrayList<Student> as =**new** ArrayList<>();

as.add(**new** Student("李四1", 11));

as.add(**new** Student("李四2", 12));

as.add(**new** Student("李四3", 13));

as.add(**new** Student("李四4", 14));

ArrayList<Worker> aw =**new** ArrayList<>();

aw.add(**new** Worker("王五1", 11));

aw.add(**new** Worker("王五2", 12));

aw.add(**new** Worker("王五3", 13));

aw.add(**new** Worker("王五4", 14));

ArrayList<Teacher> at =**new** ArrayList<>();

at.add(**new** Teacher("赵六1", 11));

at.add(**new** Teacher("赵六2", 12));

at.add(**new** Teacher("赵六3", 13));

at.add(**new** Teacher("赵六4", 14));

//1、如果show方法中的形参是Collection<?>,下述四种情况编译均通过

*show*(ap);

*show*(as);

*show*(aw);

*show*(at);

//2、如果要求show方法只能够接收Worker或者是Worker的子类

//则show方法中的形参为Collection<? extends Worker>

*show*(ap); //编译不通过

*show*(as); //编译不通过

*show*(aw);

*show*(at);

//3、如果要求show方法只能够接收Worker或者是Worker的父类

//则show方法中的形参为Collection<? super Worker>

*show*(ap);

*show*(as); //编译不通过

*show*(aw);

*show*(at); //编译不通过

}

}

### 泛型方法

**import** java.util.ArrayList;

**import** java.util.Collection;

**public** **class** GenericMethodTest {

//将一个数组中的数据存储到集合当中

**public** **static** **void** save(Object[] arr,Collection<Object> coll)

{

**for**(Object o : arr)

{

coll.add(o);

}

}

//使用泛型方法实现上述功能

**public** **static** <T> **void** saveGeneric(T[] arr,Collection<T> coll)

{

**for**(T t : arr)

{

coll.add(t);

}

}

**public** **static** **void** main(String[] args) {

//测试

Object[] oa = **new** Object[100];

Collection<Object> co = **new** ArrayList<>();

*save*(oa,co);

*saveGeneric*(oa,co); //此时T类型为Object类型

String[] sa = **new** String[100];

Collection<String> cs = **new** ArrayList<>();

// save(sa,cs); 编译出错

*saveGeneric*(sa,cs); //此时T类型为String类型

Integer [] ia =**new** Integer[100];

Float [] fa =**new** Float[100];

Number [] na =**new** Number[100];

Collection<Number> cn =**new** ArrayList<>();

//下述T类型为Number类型，分别从Integer、Float、Number-->Number

*saveGeneric*(ia,cn);

*saveGeneric*(fa,cn);

*saveGeneric*(na,cn);

//下述T类型为Object类型，从Number-->Object

*saveGeneric*(na,co);

//下述代码报错，此时T类型为String类型，Number类型并不是String的子类，因而无法转化

*saveGeneric*(na,cs);

}

}

### 泛型擦除

Java中的泛型是假泛型 ，仅仅是在编译期有效，在运行期是不存在的。

当把一个具有泛型信息的对象赋值给另外一个没有泛型信息的变量时候，所有在尖括号之间的类型信息都将被扔掉。

18.异常

异常机制已经成为判断一门编程语言是否成熟的标准，除了传统c语言没有提供异常机制，目前主流的语言，Java，C#，Ruby，Python这些都提供了完整的异常机制。

Java中处理异常主要依赖于try，catch，finally，trow和throws五个关键字。

Try语句试图捕获异常，在{}范围内进行捕获异常。

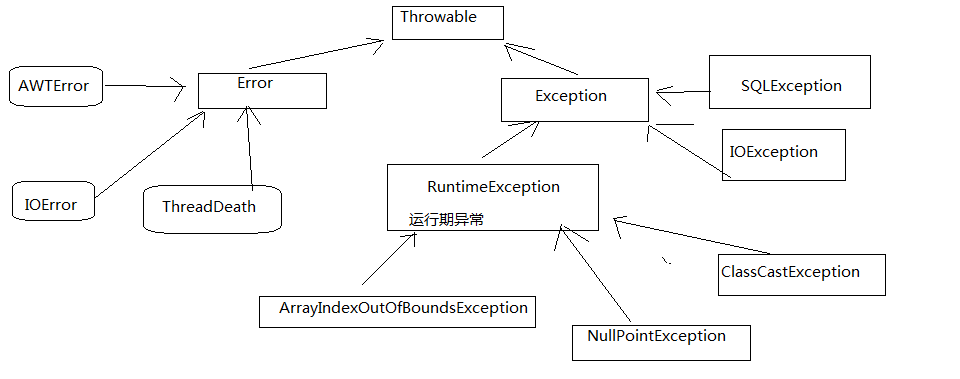
Catch用来捕获到异常后如果处理

Finnally 是无论如何都会执行代码块

Throw 和throws 是抛出异常

### 异常处理机制

Java的异常机制能让程序具有良好的容错性，让程序更加健壮，当程序出现意外的情况，系统会自动生成一个Exception对象通知程序进程处理，从而实现业务功能和实现代码的分离。



### 自定义异常类

**public** **class** SelfException **extends** RuntimeException{

/\*\*

\* 自定义异常类

\* 1、继承RuntimeException或者Exception

\* 2、写一个无参的构造函数

\* 3、写一个String类型的构造函数

\*/

**public** SelfException()

{

}

**public** SelfException(String msg)

{

**super**(msg);

}

}

**public** **class** SelfExceptionDemo {

//定义方法f1

**public** **static** **double** f1(**int** a,**int** b)

{

//求解两个数相除，要求b不能为负数，否则抛出异常

**if**(b<0)

**throw** **new** SelfException("被除数不能为负数！！");

**else** **if**(b==0)

**throw** **new** RuntimeException("被除数不能为0");

**return** a/b;

}

**public** **static** **void** main(String[] args) {

//测试

System.***out***.println(*f1*(2,5));

}

}

### 常见的RuntimeException

//测试常见的运行时异常

**public** **class** RuntimeExceptionTest {

**public** **static** **void** main(String[] args) {

//1.java.lang.NullPointerException-->空指针异常：调用了未经初始化的对象或者是不存在的对象

//数组的初始化是指对数组分配需要的空间，而初始化后的数组对应的数组元素并没有被初始化（实例化），

//其依然还是空的，所以调用的时候出现空指针异常

**int**[] arr = **null**;

System.***out***.println(arr[0]);

//2.java.lang.ClassNotFoundException-->指定的类不存在

//3. java.lang.ArithmeticException: / by zero -->数学运算异常

System.***out***.println(1/0);

//4.java.lang.ArrayIndexOutOfBoundsException--> 数组下标越界异常

String[] s = **new** String[3];

System.***out***.println(s[3]);

//5.java.lang.lllegalArgumentException-->方法参数错误异常

//调用类库的方法时出现参数传递错误

//6.java.lang.lllegalAccessException-->没有访问权限

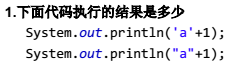
//当应用程序要调用一个类，但当前的方法没有对该类的访问权限，则会出现这个异常

}

}

19.IO操作

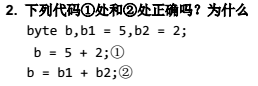
## 笔试题整理



答：

第一个输出结果为98，’a’是字符，会自动转化成相应的ASCII码值（97）后加1，则最终运算的结果“98”为输出的结果

第二个输出结果为a1，”a”本身是字符串形式，此时+（加号）对字符串起着连接作用，因此将字符串”a”与数字1连接在一起，从而组成新的字符串“a1”作为最终输出的结果



答：

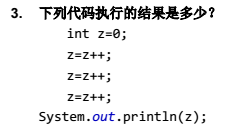
Java的自动类型转化新增了如下规则

* 所有的byte类型 short类型 和char类型 自动提升到int类型
* 整个算术表达式数据类型提升到与表达式中最高的操作数相同。

b是byte字节类型，5+2运算后的结果为7，属于byte字节范围，不会报错

b1、b2都是byte字节类型，计算之后的结果会自动提升到int型，因此需要通过强制类型转化，将计算后的结果强制转化为byte类型

结果：一处正确、二处错误，因为两个 byte数据类型相加是不确定的所以需要类型转换



答：

Int z = 0; //初始化z的值为0

z = z ++ ; //右侧的z先将值赋给左侧的z，z=0

z = z ++ ; //右侧的z先将值赋给左侧的z，z=0

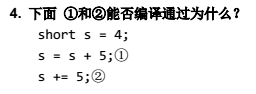
z = z ++ ; //右侧的z先将值赋给左侧的z，z=0

Int z = 0; //初始化z的值为0

z = ++ z ; //右侧的z先将值赋给左侧的z，z=1

z = ++ z ; //右侧的z先将值赋给左侧的z，z=2

z = ++ z ; //右侧的z先将值赋给左侧的z，z=3



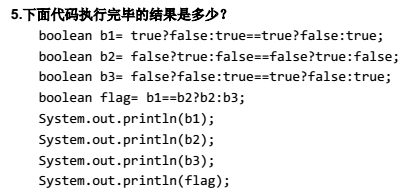
答：

**short** s = 4;

s = (**short**) (s+5); //计算之后short类型自动提升到int型需要进行强制类型转化

s+=5;

一处编译失败，整形的默认类型是 int,一个 short 类型和 int 类型相加得到是 int 所以一处需要强制类型转换，二出因为是赋值运算符能自动完成类型转换所以正确。



答：

**boolean** b1 = **true**?**false**:**true**==**true**?**false**:**true**;

**boolean** b2 = **false**?**true**:**false**==**false**?**true**:**false**;

**boolean** b3 = **false**?**false**:**true**==**true**?**false**:**true**;

**boolean** flag = b1==b2?b2:b3;

System.***out***.println(b1);

System.***out***.println(b2);

System.***out***.println(b3);

System.***out***.println(flag);

**(a)==优先与三目运算符？： (b)有多个三目运算符从右侧执行**

**等于运算符（左结合）的优先级高于条件运算符（右结合）的优先级**

**条件表达式的应用：条件?语句1：语句2**

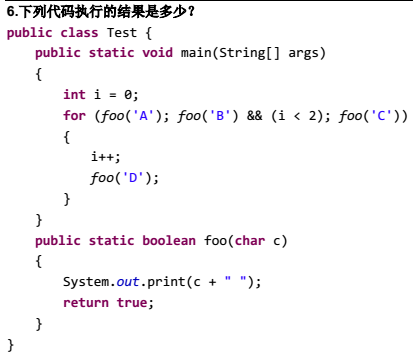
**如果条件成立则执行语句1，否则执行语句2，依次进行分析**

B1:先将true==true进行计算，进一步简化为“true?false:true?false:true”，由于条件运算符是右结合，先算后面的部分，进一步简化为“true?false:false”-->false,依次类推，下述亦是如此进行分析，一定要注意优先级的顺序，否则容易出错！！

B2:化简为“false?true:true”-->true

B3:化简为“false?false:false”-->false

Flag:化简为“false?true:false”-->false



**public** **static** **boolean** f(**char** c)

{

System.***out***.println(c);

**return** **true**;

}

**public** **static** **void** main(String[] args) {

//简单测试

// System.out.println(f(1));

**int** i=0;

**for**(*f*('A');*f*('B')&&(i<2);*f*('C'))

{

i++;

*f*('D');

}

}

答：

for（初始化条件;条件表达式;循环控制条件）

{

循环体

}

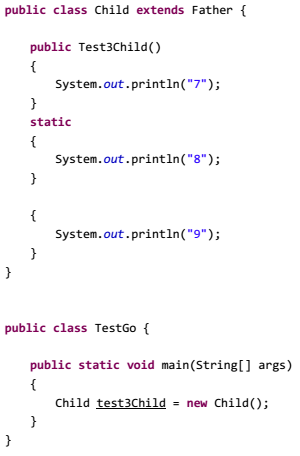
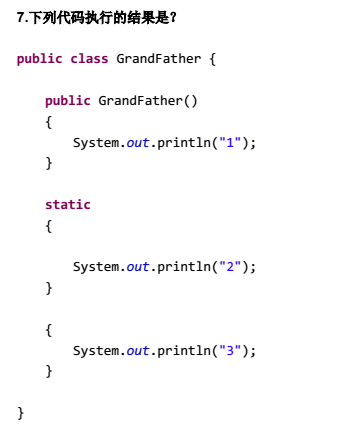
1、初始化条件只初始化一次

2、在满足条件表达式的情况下执行循环语句

3、循环控制条件

4、依次循环2、3步骤

输出结果为“A B D C B D C B”



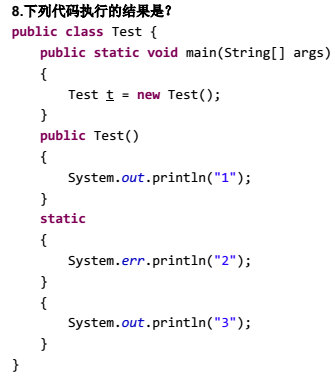
答：

代码块是在类初始化的时候自动执行的，其优先于构造函数执行

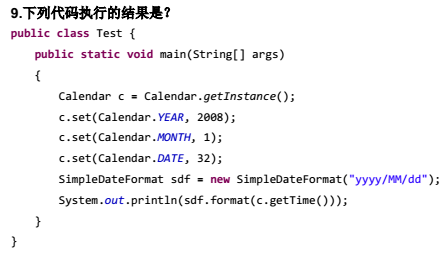
* static初始化块则是在类加载的时候自动执行1次，若同一个java类中出现多个static初始化块则按照出现的先后顺序执行（只执行1次）
* 非static的初始化块语句会在类的对象创建的时候被调用1次
* 构造方法是在类对象被创建的时候执行
* 类加载的时候执行static语句块（只执行一次），对象创建的时候先执行非static语句块后执行构造方法，对象创建一次执行一次

程序执行的时候需要创建Child对象，则程序依次加载GrandFather、Father、Child类，其相应static初始化块依次执行，输出“2 5 8”，随后依次执行每个类中的非static初始化块、初始化构造方法，分别依次输出“31 64 97”

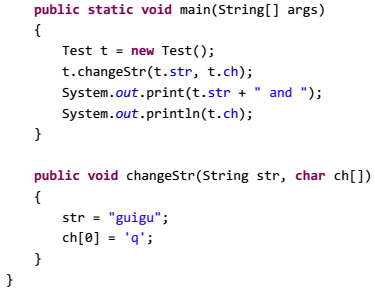
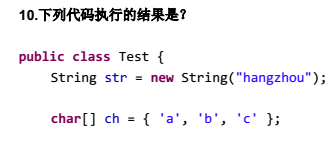
输出结果为“258 31 64 97”



答：静态代码块执行优先，随后分别是代码块、构造器，预期输出结果为“231”，但执行结果2不确定，但是3一定在1的前方 。因为2是以err输出，err是带有缓冲的输出。



答：输出的是指定格式的日期与时间，2008年是闰年，2月份有29天，则输出日期为“2008/3/3”



答：

**package** com.guigu.vocation.hw;

**public** **class** Test1 {

String str = **new** String("hangzhou");

**char**[] ch = {'a','b','c'};

**public** **void** changeStr(String str,**char** ch[])

{

str = "guigu";

ch[0] = 'q';

}

**public** **static** **void** main(String[] args) {

Test1 t = **new** Test1();

t.changeStr(t.str,t.ch);

System.***out***.print(t.str+"and");

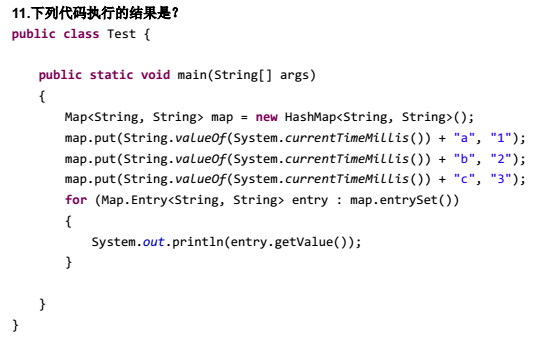
System.***out***.println(t.ch);

}

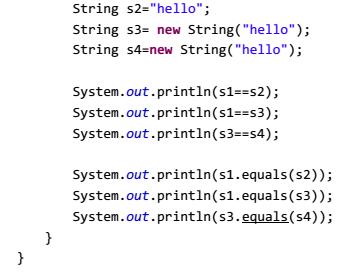
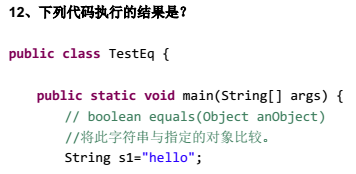
}

输出结果：hangzhouandqbc，数组等引用类型发生改变，基本类型、String类型不发生改变

即：八种基本类型不会发生变化，抽象类型中除了String类型其余都会发生改变



答：HashMap无法保证一定的访问次序，因此输出的结果次序每次都不一定一致



答：

内存的五片区域 ：寄存器，栈内存，堆内存，方法区，本地方法区

栈内存：存储的都是局部变量，一旦使用完毕，立即释放

堆内存：存储的是实体(对象)

Java程序中有两种方式是两个变量是否相等。一种是利用==判断，另外一种是使用equals方法来判断。

当使用==进行判断的时候，如果两个变量都是基本类型的变量，并且都是数值类型，则只要两个变量的值相同时，便返回true。但如果两个变量的类型是引用类型，则只有两个变量均指向相同的对象才返回true。

使用equals方法进行判断时则以具体equals方法中实际比较的项目为准（使用equals判断两个字符串是否相等），可通过重写equals方法实现自定义的比较方式

**package** com.guigu.vocation.hw;

**public** **class** TestEq {

**public** **static** **void** main(String[] args) {

//boolean equals(Object anObject)

//将此字符串与指定的对象进行比较

String s1 = "hello"; //已经确定下来的字符串定义在常量池中

String s2 = "hello"; //已经确定下来的字符串定义在常量池中

String s3 = **new** String("hello"); //通过new开辟新的空间

String s4 = **new** String("hello"); //通过new开辟新的空间

System.***out***.println(s1==s2); //true

System.***out***.println(s1==s3); //false

System.***out***.println(s3==s4); //false

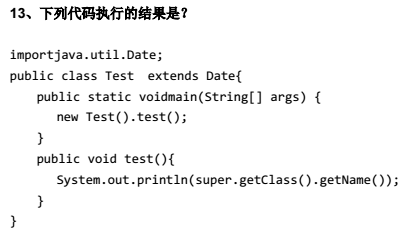
System.***out***.println(s1.equals(s2)); //true

System.***out***.println(s1.equals(s3)); //true

System.***out***.println(s3.equals(s4)); //true

}

}



答：

**package** com.guigu.vocation.hw;

**import** java.util.Date;

**public** **class** Test **extends** Date{

**public** **static** **void** main(String[] args) {

**new** Test().test();

}

**public** **void** test(){

System.***out***.println(**super**.getClass().getName());

}

}

输出的是Test对象的类名，在test方法中，直接调用getClass().getName()方法，返回的是Test类名

由于getClass()在Object类中定义成了final，子类不能覆盖该方法，所以，在 test方法中调用getClass().getName()方法，其实就是在调用从父类继承的getClass() 方法，等效于调用super.getClass().getName()方法，所以， super.getClass().getName()方法返回的也应该是Test。 如果想得到父类的名称，应该用如下代码： getClass().getSuperClass().getName();